EXAMEN 2ºEVA. POO

COLECCIONES:

Las colecciones son estructuras que permiten almacenar, procesar y gestionar datos de manera eficiente. Para trabajar con colecciones, Java ofrece la API de *Collections Framework*, que incluye interfaces, clases y algoritmos. Aquí tienes un resumen:

Principales Interfaces de Colecciones:

1. **List**: Almacena elementos ordenados y permite duplicados. Ejemplos:
   - `ArrayList`
   - `LinkedList`
2. **Set**: No permite duplicados. Ejemplos:
   - `HashSet`
   - `TreeSet`
   - `LinkedHashSet`
3. **Map**: Almacena pares clave-valor, donde cada clave es única. Ejemplos:
   - `HashMap`
   - `TreeMap`
   - `LinkedHashMap`
4. **Queue**: Sigue una estructura de datos FIFO (First In, First Out). Ejemplos:
   - `PriorityQueue`
   - `Deque` (doble cola)

```java
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Crear una colección de tipo List
        List<String> frutas = new ArrayList<>();

        // Añadir elementos
        frutas.add("Manzana");
        frutas.add("Plátano");
        frutas.add("Cereza");

        // Iterar sobre la colección
        for (String fruta : frutas) {
            System.out.println(fruta);
        }
    }
}
```

IMPLEMENTAR COLECCIONES:

a. Crea una clase que represente un objeto:

```java
public class Persona {
    private String nombre;
    private int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public String getNombre() {
        return nombre;
    }

    public int getEdad() {
        return edad;
    }

    @Override
    public String toString() {
        return "Nombre: " + nombre + ", Edad: " + edad;
    }
}
```

b. Utiliza una colección para gestionar múltiples instancias de esta clase:

```java
import java.util.ArrayList;
import java.util.List;

public class EjemploColecciones {
    public static void main(String[] args) {
        List<Persona> personas = new ArrayList<>();

        personas.add(new Persona("Carlos", 30));
        personas.add(new Persona("Ana", 25));
        personas.add(new Persona("Luis", 35));

        for (Persona persona : personas) {
            System.out.println(persona);
        }
    }
}
```

Ventajas de las colecciones en POO:

- **Encapsulación**: Puedes mantener la lógica de manejo de datos dentro de métodos.
- **Flexibilidad**: Cambiar el tipo de colección sin afectar el resto del código.
- **Reutilización**: Utilizar clases genéricas y polimorfismo.

INTERFACES:

Una **interfaz** es una especie de "contrato" que define un conjunto de métodos que una clase debe implementar. Esto fomenta el desacoplamiento y facilita la reutilización y flexibilidad del código.

¿Qué es una interfaz?

- Es una estructura que contiene **métodos abstractos** (sin cuerpo) y/o **constantes**.
- Las clases que implementan una interfaz **deben proporcionar una implementación para todos los métodos abstractos** que contiene.

Por ejemplo:

```java
public interface Animal {
    void comer();
    void dormir();
}
```

En este caso, cualquier clase que implemente la interfaz `Animal` tendrá que proporcionar las implementaciones de los métodos `comer()` y `dormir()`.

Implementación de una interfaz

Supongamos que queremos modelar un sistema donde diferentes tipos de animales comparten ciertas características.

```java
public class Perro implements Animal {
    @Override
    public void comer() {
        System.out.println("El perro está comiendo.");
    }

    @Override
    public void dormir() {
        System.out.println("El perro está durmiendo.");
    }
}
```

Ventajas de usar interfaces

1. **Polimorfismo**: Puedes tratar diferentes clases de manera uniforme si implementan la misma interfaz.

```java
public class TestAnimal {
    public static void main(String[] args) {
        Animal perro = new Perro();
        Animal gato = new Gato();

        perro.comer();
        gato.dormir();
    }
}
```

2. **Herencia múltiple simulada**: Una clase en Java puede implementar múltiples interfaces, aunque no puede heredar de varias clases. Ejemplo:

```java
public interface Volador {
    void volar();
}

public class Murcielago implements Animal, Volador {
    @Override
    public void comer() {
        System.out.println("El murciélago está comiendo.");
    }

    @Override
    public void dormir() {
        System.out.println("El murciélago está durmiendo.");
    }

    @Override
    public void volar() {
        System.out.println("El murciélago está volando.");
    }
}
```

Métodos `default` y `static` en interfaces (Java 8+)

A partir de Java 8, las interfaces también pueden tener métodos con cuerpo si se declaran como `default` o `static`.

Ejemplo:

```java
public interface Animal {
    void comer();
    void dormir();

    default void sonido() {
        System.out.println("Algunos animales hacen sonidos.");
    }
}
```

Una clase que implemente esta interfaz puede usar este método `default` directamente o sobreescribirlo.

HERENCIA:

La **herencia** en la programación orientada a objetos (POO) es un mecanismo que permite a una clase (llamada **subclase** o **clase hija**) adquirir las propiedades y comportamientos (campos y métodos) de otra clase (llamada **superclase** o **clase padre**). Esto fomenta la reutilización de código y la creación de jerarquías lógicas.

En **Java**, la herencia se implementa con la palabra clave `extends`.

Conceptos principales:

1. **Superclase**:
    - Es la clase base que contiene atributos y métodos comunes.
    - Las subclases heredan estos elementos.
2. **Subclase**:
    - Es la clase que extiende (hereda) de la superclase.
    - Puede agregar nuevos atributos y métodos, o sobrescribir los métodos heredados.
3. **Palabra clave** `extends`:
    - Se utiliza para declarar que una clase hereda de otra.

```
public class Animal {
    // Superclase
}

public class Perro extends Animal {
    // Subclase
}
```

Modificadores de acceso:

- La herencia está influida por los niveles de acceso (`public`, `protected`, `private`).
- Los miembros `private` de la superclase no son accesibles directamente en la subclase.

Ejemplo práctico

Supongamos que tenemos un sistema para gestionar animales:

Superclase:

```
public class Animal {
    protected String nombre;

    public Animal(String nombre) {
        this.nombre = nombre;
    }

    public void comer() {
        System.out.println(nombre + " está comiendo.");
    }

    public void dormir() {
        System.out.println(nombre + " está durmiendo.");
    }
}
```

Subclase:

```java
public class Perro extends Animal {
    private String raza;

    public Perro(String nombre, String raza) {
        super(nombre); // Llama al constructor de la superclase
        this.raza = raza;
    }

    public void ladrar() {
        System.out.println(nombre + " está ladrando.");
    }

    @Override
    public void comer() {
        System.out.println(nombre + " (un perro de raza " + raza + ") está
    }
}
```

Clase principal:

```java
public class HerenciaEjemplo {
    public static void main(String[] args) {
        Perro miPerro = new Perro("Max", "Labrador");

        miPerro.comer();   // Sobrescribe el método de la superclase
        miPerro.dormir();  // Llamado directamente desde la superclase
        miPerro.ladrar();  // Método específico de la subclase
    }
}
```

Características clave:

1. **Reutilización de código:**
    - Los atributos y métodos comunes se definen en la superclase, evitando duplicación.
2. **Polimorfismo:**
    - Una subclase puede ser tratada como una instancia de la superclase.

```java
Animal animal = new Perro("Rex", "Beagle");
animal.comer(); // Ejecuta el método sobrescrito en la subclase
```

Sobrescritura (`@Override`):

- Las subclases pueden redefinir métodos heredados para cambiar su comportamiento.

Clase `Object`:

- Todas las clases en Java heredan implícitamente de `Object`, la superclase raíz.

Limitaciones de la herencia:

1. Java **no soporta herencia múltiple** directa (una clase no puede extender más de una clase) para evitar ambigüedades. Sin embargo, esto se puede lograr usando **interfaces**.
2. **Acoplamiento**:
    - Un uso excesivo de herencia puede hacer que el sistema sea rígido y difícil de mantener.

TIPOS ENUMERADOS, CONSTANTES Y MATRICES:

Limitaciones de la herencia:

1. Java **no soporta herencia múltiple** directa (una clase no puede extender más de una clase) para evitar ambigüedades. Sin embargo, esto se puede lograr usando **interfaces**.
2. **Acoplamiento**:
    - Un uso excesivo de herencia puede hacer que el sistema sea rígido y difícil de mantener.

```
public enum Dia {
    LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO;
}
```

Uso de un enum:

```
public class EjemploEnum {
    public static void main(String[] args) {
        Dia hoy = Dia.VIERNES;

        switch (hoy) {
            case LUNES:
                System.out.println("¡Ánimo, empieza la semana!");
                break;
            case VIERNES:
                System.out.println("¡Es viernes, el cuerpo lo sabe!");
                break;
            default:
                System.out.println("Es un día más.");
        }
    }
}
```

Ventajas de los enums:

- Facilitan la lectura y el mantenimiento del código.
- Garantizan que los valores sean constantes y limitados.

- Se pueden asociar métodos, constructores y variables.

Enum con métodos:

```java
public enum Estado {
    INICIADO, EN_PROGRESO, COMPLETADO;

    public boolean esFinalizado() {
        return this == COMPLETADO;
    }
}
```

2. Constantes

Una **constante** es una variable cuyo valor no cambia durante la ejecución del programa. En Java, las constantes se declaran usando la palabra clave `final`.

Declaración de constantes:

```java
public class Constantes {
    public static final double PI = 3.14159;
    public static final String MENSAJE = "Hola, mundo!";
}
```

Uso de constantes:

```java
public class EjemploConstantes {
    public static void main(String[] args) {
        System.out.println("El valor de PI es: " + Constantes.PI);
        System.out.println(Constantes.MENSAJE);
    }
}
```

Características:

- `final`: Evita que la constante se modifique después de inicializarla.
- `static`: Permite acceder a la constante sin crear una instancia de la clase.

Ventajas:

- Facilita el mantenimiento del código al centralizar valores.
- Mejora la legibilidad al usar nombres significativos en lugar de valores mágicos (hard-coded).

3. Matrices (Arrays)

Una **matriz** es una estructura de datos que almacena múltiples valores del mismo tipo en un espacio continuo de memoria. Es útil para representar colecciones de datos en POO.

Declaración de una matriz:

```java
int[] numeros = new int[5]; // Matriz de 5 elementos
```

Inicialización de una matriz:

```java
int[] numeros = {1, 2, 3, 4, 5};
```

Iteración sobre una matriz:

```java
public class EjemploMatriz {
    public static void main(String[] args) {
        int[] numeros = {10, 20, 30, 40, 50};

        for (int i = 0; i < numeros.length; i++) {
            System.out.println("Elemento en posición " + i + ": " + numeros[i]);
        }
    }
}
```

Matrices multidimensionales:

```java
public class MatrizMultidimensional {
    public static void main(String[] args) {
        int[][] matriz = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        for (int i = 0; i < matriz.length; i++) {
            for (int j = 0; j < matriz[i].length; j++) {
                System.out.print(matriz[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Características:

- Longitud fija: Una vez definida, no puedes cambiar el tamaño de la matriz.
- Los elementos son accesibles por índice: `matriz[i]`.

Ventajas:

- Permiten almacenar grandes cantidades de datos de manera estructurada.
- Son rápidas en términos de acceso y manipulación.

EXCEPCIONES Y TRY CATCH:

Las excepciones en Java son un mecanismo para manejar errores o situaciones excepcionales que ocurren durante la ejecución de un programa. En la programación orientada a objetos (POO), las excepciones se utilizan para garantizar que el programa pueda responder de manera controlada a errores inesperados y continuar ejecutándose sin fallar por completo.

El uso de bloques try-catch es una técnica clave para capturar y manejar estas excepciones.

Conceptos clave:

1. **Excepción**:
   - Un evento anómalo que interrumpe el flujo normal del programa.
   - Ejemplos: división por cero, acceso a un índice fuera de los límites de una matriz, archivo no encontrado.
2. `try`:
   - Define un bloque de código que se desea ejecutar de forma segura.
   - Si ocurre una excepción dentro del bloque, se transfiere al bloque `catch`.
3. `catch`:
   - Captura y maneja la excepción lanzada dentro del bloque `try`.
4. `finally` **(opcional)**:
   - Define un bloque de código que se ejecuta siempre, sin importar si hubo o no una excepción.
   - Útil para liberar recursos (cerrar archivos, conexiones, etc.).

Sintaxis básica:

```
try {
    // Código que puede generar una excepción
} catch (TipoDeExcepcion e) {
    // Código para manejar la excepción
} finally {
    // Código que siempre se ejecuta (opcional)
}
```

Ejemplo práctico:

Supongamos que queremos manejar una excepción de división por cero.

```java
public class EjemploTryCatch {
    public static void main(String[] args) {
        try {
            int numerador = 10;
            int denominador = 0;

            // Esta línea genera una excepción: ArithmeticException
            int resultado = numerador / denominador;

            System.out.println("Resultado: " + resultado);
        } catch (ArithmeticException e) {
            // Manejo de la excepción
            System.out.println("Error: No se puede dividir entre cero.");
        } finally {
            // Código que siempre se ejecuta
            System.out.println("Finalizando la operación.");
        }
    }
}
```

Manejo de múltiples excepciones:

Un programa puede generar diferentes tipos de excepciones, y puedes manejar cada tipo de forma específica:

```java
public class MultiplesExcepciones {
    public static void main(String[] args) {
        try {
            int[] numeros = {1, 2, 3};
            System.out.println(numeros[5]); // Genera ArrayIndexOutOfBoundsI
        } catch (ArithmeticException e) {
            System.out.println("Error aritmético: " + e.getMessage());
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error de índice: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Ocurrió un error general: " + e.getMessage(
        }
    }
}
```

Excepciones personalizadas:

Puedes crear tus propias excepciones al extender la clase `Exception` o `RuntimeException`.

```java
public class MiExcepcion extends Exception {
    public MiExcepcion(String mensaje) {
        super(mensaje);
    }
}

public class TestExcepcion {
    public static void main(String[] args) {
        try {
            lanzarExcepcion();
        } catch (MiExcepcion e) {
            System.out.println("Capturada: " + e.getMessage());
        }
    }

    public static void lanzarExcepcion() throws MiExcepcion {
        throw new MiExcepcion("Esto es una excepción personalizada.");
    }
}
```

DIAGRAMA UML:

Los **diagramas UML** (Lenguaje Unificado de Modelado, por sus siglas en inglés) son una herramienta clave en la programación orientada a objetos (POO) y se utilizan para diseñar y visualizar cómo se organizan las clases, los objetos, y las relaciones entre ellos en un sistema. Aunque no son exclusivos de Java, son muy útiles para modelar proyectos en este lenguaje.

Diagrama de Clases:

- Muestra las clases del sistema, sus atributos, métodos y las relaciones entre ellas (como herencia, asociación o composición).
- Es el más importante en POO, ya que define la estructura del sistema.

En este ejemplo:

- `Persona` es la clase padre.
- `Estudiante` hereda de `Persona`.

```java
public class Persona {
    private String nombre;
    private int edad;

    public String getNombre() {
        return nombre;
    }

    public int getEdad() {
        return edad;
    }
}

public class Estudiante extends Persona {
    private String matricula;

    public String getMatricula() {
        return matricula;
    }
}
```

ABSTRACCIÓN:

La abstracción consiste en ocultar los detalles complejos del funcionamiento interno de un objeto y mostrar solo las funcionalidades esenciales. Esto permite centrarse en el *qué hace* un objeto en lugar de *cómo lo hace*. En Java, se implementa mediante **clases abstractas** e **interfaces**.

- **Clase abstracta**: Define métodos que deben ser implementados por las clases derivadas.
- **Interfaz**: Establece un contrato que las clases que la implementan deben cumplir.

Beneficio: Ayuda a reducir la complejidad al exponer sólo lo necesario.

```java
abstract class Animal {
    abstract void sonido();
}

class Perro extends Animal {
    void sonido() {
        System.out.println("El perro ladra.");
    }
}
```

ENCAPSULACIÓN:

La encapsulación consiste en proteger los datos de una clase al restringir el acceso directo a sus atributos (variables) y proporcionar métodos públicos (**getters y setters**) para controlarlo. Esto se logra mediante modificadores de acceso como `private`, `protected`, y `public`.

- **Atributos privados**: No son accesibles directamente desde fuera de la clase.
- **Métodos públicos**: Permiten acceder y modificar esos atributos de manera controlada.

Beneficio: Protege los datos de la manipulación indebida y mejora la seguridad del código.

```java
class Persona {
    private String nombre;

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

POLIMORFISMO:

El polimorfismo permite que un objeto adopte diferentes formas. Es decir, un mismo método puede comportarse de manera diferente dependiendo del contexto o del tipo de objeto que lo invoque. En Java, se implementa principalmente mediante **sobrescritura** y **sobrecarga de métodos**.

- **Sobrescritura**: Una subclase redefine el comportamiento de un método heredado de la superclase.
- **Sobrecarga**: Un método tiene el mismo nombre pero diferentes parámetros dentro de la misma clase.

Beneficio: Fomenta la flexibilidad y la reutilización de código.

```java
class Animal {
    void sonido() {
        System.out.println("Algún sonido.");
    }
}

class Gato extends Animal {
    void sonido() {
        System.out.println("El gato maúlla.");
    }
}
```

RETURN CON COMPARETO:

El método `compareTo()` es parte de la interfaz `Comparable`, y se utiliza para comparar objetos. Este método retorna un **valor entero** que indica la relación entre el objeto actual y el objeto comparado. Es fundamental para realizar comparaciones personalizadas en clases de POO, como en ordenamientos o búsquedas.

¿Qué devuelve `compareTo()`?

- **Valor negativo**: Si el objeto actual es **menor** que el objeto comparado.
- **Cero (0)**: Si los dos objetos son **iguales**.
- **Valor positivo**: Si el objeto actual es **mayor** que el objeto comparado.

Ejemplo práctico:

Supongamos que queremos comparar objetos de una clase `Persona` por su nombre:

```java
public class Persona implements Comparable<Persona> {
    private String nombre;

    public Persona(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }

    // Implementación de compareTo
    @Override
    public int compareTo(Persona otraPersona) {
        return this.nombre.compareTo(otraPersona.getNombre());
    }
}
```

TIPOS GENÉRICOS:

Genéricos: Son una herramienta en Java que permite que clases, interfaces y métodos operen con cualquier tipo de datos definido en tiempo de compilación. Utilizan un marcador de tipo como `T`, y su propósito es ofrecer flexibilidad, reutilización de código y seguridad de tipos al evitar errores en tiempo de ejecución.

Encapsulación: Es el principio de POO que protege los datos dentro de una clase al restringir el acceso directo a sus atributos. Esto se logra con modificadores de acceso como `private` y el uso de métodos públicos (`getters` y `setters`) para acceder o modificar los datos, asegurando que estos estén controlados y protegidos contra modificaciones indebidas.

Polimorfismo: Es la capacidad de un objeto de adoptar diferentes formas y comportarse de manera distinta según el contexto. En Java, se implementa principalmente mediante la sobrescritura de métodos (donde una subclase redefine el comportamiento de un método heredado) y la sobrecarga de métodos (donde un método tiene el mismo nombre pero diferentes parámetros).

CompareTo: Es un método que forma parte de la interfaz `Comparable` y se utiliza para comparar objetos en Java. Devuelve un número entero que indica si el objeto actual es menor, igual o mayor que el objeto con el que se está comparando. Es útil para ordenar colecciones y personalizar los criterios de comparación.

```java
public class Caja<T extends Comparable<T>> {
    private T elemento;

    public void guardar(T item) { this.elemento = item; }
    public T obtener() { return elemento; }
}

class Producto implements Comparable<Producto> {
    private String nombre;
    private double precio;

    public Producto(String nombre, double precio) { this.nombre = nombre; tl

    @Override
    public int compareTo(Producto otro) {
        return Double.compare(this.precio, otro.precio);
    }

    @Override
    public String toString() { return nombre + " ($" + precio + ")"; }
}

// Uso
public class Main {
    public static void main(String[] args) {
        Caja<Producto> caja = new Caja<>();
        caja.guardar(new Producto("Laptop", 1000.0));
        System.out.println(caja.obtener());
    }
}
```